Face Detection and Recognition

Michael Lane
Portland State University
1825 SW Broadway
Portland, OR 97201
lane7@pdx.edu

Abstract

Face detection and recognition has now reached the status of a classical problem in Computer Vision. There are many different ways of recognizing and then classifying digital images of faces. This project implements the Haar Face Cascade algorithm for face detection. For face recognition OpenFace, seven different classifiers from Scikit-Learn *(all of which were trained on the LFW data set), and Amazon's AWS Rekognition were implemented.*

1. Introduction

Face detection and recognition in digital images has reached the status of a classic Computer Vision problem. Most humans can detect faces and recognize a known face with surprising ease and accuracy, so it seems that in order to build an Artificial (human) Intelligence, face detection and recognition is a basic requirement. Additionally, it would be nice to have an accurate and efficient face detection and recognition system for things like security systems, law enforcement, *etc*.

Older face detection algorithms such as *Eigenfaces* [10] tended to utilize a statistical and linear algebra analysis on a group of faces to determine a set of abstracted features for a given individual face. This *eigenface* can then be used as the basis of other classifiers for new faces such as a Support Vector Machine.

More recently, researchers have utilized some hand crafted feature detectors that are commonly known as a Haar Cascade, so named because of the use of detectors that are similar to Haar Wavelets [11]. On a high level, these classifiers work by exhaustively searching an image for a feature common to human faces that is easily identified by a Haar Wavelet. For example, the bridge of the nose might be identified by a long wavelet that is white sandwiched with black since the bridge of the nose is generally lighter than the areas to either side of it. Similarly the eyes are usually darker than the cheeks, and therefore a wavelet with a dark area above a light area would be able to detect this feature. A Haar Cascade Detector was implemented for the face detection experiment in Section 2. What's more, the face recognition classifiers I've implemented in section 3.3 utilize eigenfaces via Primary Components Analysis (PCA) as the inputs to the various classifiers.

The current state-of-the-art in object and face detection and recognition is deep Convolutional Neural Networks (CNNs). CNNs have proven themselves to be quite good at just about any image recognition task and face recognition in particular [6]. How CNNs work is a matter of active research; however, current research using Adversarial Generative Networks indicates that the nodes in the fully connected network act as extremely complex feature detectors that specialize in identifying certain aspects of the image [7]. Lower layers specialize in tasks like corner and line detection. Middle layers specialize in tasks like eye detection. Top layers specialize in whole-face recognition. The current thinking is that a given neuron is specialized for a class of image rather than the recognition task being spread throughout the network. The OpenFace system used in section 3.1 implements a Deep Neural Network that is trained on the Labeled Faces in the Wild data set [5].

2. Face Detection

Given the status of face detection and recognition in the arena of Computer Vision, it is hardly a surprise that well known libraries like OpenCV3 [3] have built-in face detection and recognition algorithms.

2.1. Technology Used

The technology used for this example is Python 3.6, the Python environment management tool pyenv-virtualenv, OpenCV 3.2.0, the Python interface to OpenCV called cv2, Matplotlib, and a tool for quickly mocking up and running Python programs called Jupyter Notebook. The code itself is a modified version of a tutorial found online [9]. The code used for this experiment can be found at `https://goo.gl/hEScHY`

Figure 1. Abba

Figure 2. A test input image of an adult and young boys

2.2. Example Inputs

This experiment was run on three examples. Figure 1 is a photo that the Real Python tutorial used of the band Abba. The faces in this image are easily distinguishable, the background is uncluttered, and there isn't much about their bodies or clothing that would cause an issue with the face detection algorithm. One issue with this image is that it is 500×426 pixels which is much smaller than the other images used which required manual adjustments to the settings on larger images.

Figure 2 is an image of an adult and two young boys against an uncomplicated background. This was taken with a modern digital SLR, so the file was much larger: 5616×3744 pixels. Because of this, the minimum size of a bounding box was set using `minSize=(750, 750)` (whereas, it was `(30, 30)` in the Abba picture). If the `minSize` value was not changed in this way, the algorithm discovered too many faces.

Figure 3 is a strange image that a classmate shared that is designed to fool face detectors of this type.

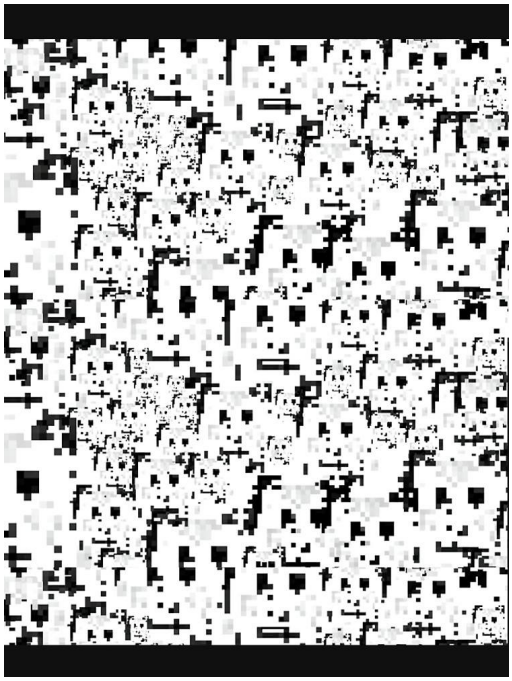

Figure 3. An "adversarial" example

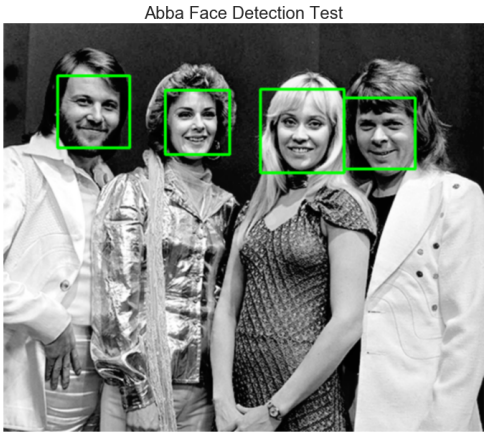

Figure 4. Abba with faces detected

2.3. Example Outputs

Figure 4 is Abba with faces detected. For this result the hyperparameters were `scaleFactor = 1.1` and `minNeighbors = 5` and the value for `minSize` that was mentioned earlier. The result is quite good with boxes right were a human would probably put them if she were doing this task manually.

Figure 5 is an adult and two young boys with faces detected. The same hyperparameters were used except for the `minSize` mentioned earlier. As you can see, the detector with the appropriate hyperparameters did a great job at

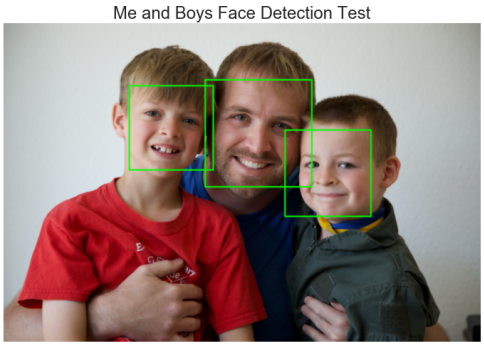

Figure 5. An adult and young boys with faces detected

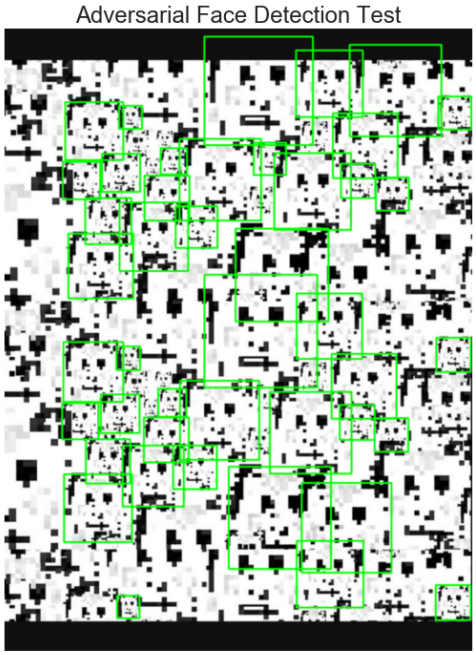

Figure 6. The adversarial example goes a bit crazy, as expected.

detecting the faces.

Figure 6 we can see that this image is intended to be a good reminder that the Haar Cascade Classifier really is not detecting faces, rather, the classifier is detecting areas of the image that have face-like shapes. The bounding box minimum size was set using `minSize=(30, 30)` for this image, and since the image is a bit larger (750×1000 pixels), this resulted in 44 "faces" being detected.

3. Face Recognition

Face detection, especially scale-invariant face detection, is arguably a more difficult task than Face Recognition. As the adversarial example in Figure 6 demonstrated, it is not always clear how to tell a computer to recognize certain groups of numbers as a face (or to reject it as not being a face). Once the faces in a given image have been identified, face recognition takes only a straight-forward machine learning classification task.

Machine learning classifiers are becoming ubiquitous these days. Most programming languages have at least one library that allows a programmer to quickly and easily train one classifier or another on a given data set. This is not to say that the development of Machine Learning was *easy*, of course, but now that these classifiers are so easy to build and run, it only takes a few minutes to train a reasonably accurate face recognition classifier.

This section explains several classifiers and discusses the results of implementing those classifiers.

3.1. OpenFace

At its heart, a CNN is a method of extracting features from an image. How CNNs work is beyond the scope of this paper; however, there are certain relevant facts to the topic at hand. There is a give-and-take with any technology. One drawback of the Haar Cascade Classifier, for example, was that it is highly dependent on the size of the image. One drawback of modern CNNs is that they require a large amount of data and computational power to train. Consumer grade laptops are generally not up to the task of training these networks on face data sets in any reasonable amount of time. The good news is that once a CNN has been trained, the weights can be stored and built in to classifiers that implement the same network structure.

This is exactly the idea behind OpenFace [2]. According to the OpenFace website:

> "OpenFace is a Python and Torch implementation of face recognition with deep neural networks and is based on the CVPR 2015 paper FaceNet: A Unified Embedding for Face Recognition and Clustering by Florian Schroff, Dmitry Kalenichenko, and James Philbin at Google. Torch allows the network to be executed on a CPU or with CUDA." [1]

One nice feature of OpenFace is that there is a Docker distribution so that the package runs as a virtual machine (that Docker calls a "container") that includes all the necessary files, CNN weights, and dependencies. So training OpenFace on your own images is as simple as installing the Docker container, running it, placing image files in the correct location, and then calling command line instructions on the network that includes your custom images.

There are some limitations to OpenFace. For one, the images can only contain a single face. It can be much more difficult than anticipated to find a picture with a single face, particularly with children or from vacation photos. Even having a picture of a face or a statue in the background could

cause problems. Another issue is that while the Docker implementation is nice, it is not trivial to implement OpenFace in such a way using the container as to make it available for a general application external to the container. Also, the OpenFace implementation uses Python 2 which means there is not necessarily an easy way to run it if your stack uses Python 3.

Nevertheless, OpenFace was, ultimately, very easy to implement for the purposes of this paper.

3.1.1 Running OpenFace

This project implemented an OpenFace tutorial [4]. The first requirement was to install docker, pull a docker image called `bamos/openface` that has everything configured, and then run that image.

Once the docker image is set up and running, copy all the training images into folders labeled for each person to be recognized. For this paper the directory structure in the Docker container was

```
/root
  /openface
    /training-images
      /joshua-lane
      /michael-lane
      /rebecca-brown
      /william-lane
```

Training images were placed in each of the corresponding person folders. See section 3.1.2 for examples of the training images. Once that is done, the faces must be detected and aligned. See Figure 7 for the code that accomplishes this task.

This will output a set of face images that have been cropped and aligned to the folder `aligned-images/`. Once this is complete, the aligned images are used to generate the embeddings. The code to execute this is found in Figure 8

The embeddings will be stored in a CSV file in the folder `./generated-embeddings/`. This CSV file will be used to train a Support Vector Machine by running the instruction in Figure 9.

This will output a trained SVM to `./generated-embeddings/classifier.pkl` which can be used to classify new examples. New sample images should be placed into the `./testing-images/` folder and it will be helpful if each testing image filename has the name of the person in the image. To run the classifier on all testing images, use the code found in Figure 10

For each of the images in the folder `./testing-images/` the classifier will run and will (usually) return a prediction and a confidence score. This projects results are discussed in section 3.1.3.

3.1.2 Example Training Images

For this paper, OpenFace was trained on four different people who are given the following labels: `michael-lane`, `rebecca-brown`, `william-lane`, and `joshua-lane`. There were between 8 and 10 training images for each person, some representative images from the training sets are found in Figures 11, 12, 13, and 14.

After the cropping and alignment phase of OpenFace, those images were changed to the images found in Figures 15, 16, 17, and 18.

3.1.3 Results

The results for OpenFace were as good as expected. The following are the predictions for a few interesting test images:

```
=== testing-images/Joshua-Lane-3.jpg ===
Predict joshua-lane with 0.80 confidence.

=== testing-images/Michael-Lane-1.jpg ===
Predict michael-lane with 0.67 confidence.

=== testing-images/Rebecca-Brown-1.jpg ===
Predict rebecca-brown with 0.85 confidence.

=== testing-images/William-Lane-4.jpg ===
```

Figures 19, 20, 21, and 22 are these interesting test images. The results were good overall, even despite face paint as in Figure 19, strange headgear as in Figure 20, or a small face covered with hair in as in Figure 21. The one image in these four examples that didn't classify was the image in Figure 22 where the face is occluded by sunglasses, hair, and dark shadows.

3.2. Amazon Rekognition

One of the newest services provided by Amazon's AWS is called "Rekognition". AWS aims to be a one-stop shop for all kinds of technology, and machine learning and computer vision is no exception. Rekognition is AWS's implementation of object and face detection and recognition. Businesses and individuals can easily implement a high quality and robust face detection and recognition in support of things like security validation, photo manipulation, personal projects and the like.

For this project, the demo web console version of AWS Rekognition found at `https://aws.amazon.com/rekognition/`. (*Note: a free AWS developer account is required to use the demo.*) Figures 23 and 24 are screen shot examples from the Rekognition demo interface. The algorithm seemed to work just about as well as OpenFace, classifying "easy" examples with high confidence.

```
cd ~/openface
./util/align-dlib.py ./training-images/ align outerEyesAndNose ./aligned-images/
 ↪  --size 96
```

Figure 7. Command line instructions used to detect and align faces in OpenFace

```
./batch-represent/main.lua -outDir ./generated-embeddings/ -data
 ↪  ./aligned-images/
```

Figure 8. Command line instructions to generate the embeddings from the aligned images

But, like OpenFace, Rekognition fails when there are occlusions and other difficulties, as in Figures 25 and 26.

AWS Rekognition also offers integrations into various programming languages via libraires. In the interest of time, those libraries were not implemented here. However, the online demo returns the results exactly as they would be returned if called via an online library. Even the JSON object is presented in the demo.

3.3. Scikit-Learn

Scikit-Learn is a machine learning library for Python that implements dozens of classifiers, regressors, supervised learning, semi-supervised learning, unsupervised learning, data set downloading, classifier evaluation, analytics tools, etc. Since the process of face recognition is a simple matter of face detection, classifier training, and then classification, Scikit-Learn is an excellent library to use to easily and quickly implement face recognition. What's more, there is an example implementation of face recognition in the Scikit-Learn documentation [8].

3.3.1 Implementation

The general process for classifying faces using Scikit-Learn is as follows:

1) Use Scikit-Learn's method `fetch_lfw_people()` to download the Labeled Faces in the Wild data set. Figure 27 shows some examples of the faces found in the LFW dataset.
2) Extract the LFW data into training and testing sets and the associated labels.
3) Use Scikit-Learn's Primary Components Analysis to extract the eigenfaces from the data.
4) Set up and run one of the Scikit-Learn classifiers and train the classifier on the training set.
5) Use the test set data to make predictions for later comparison.
6) Use Scikit-Learn's method `classification_report()` and method `confusion_matrix()` to evaluate the predictions compared to the ground truth.

This project implemented 7 different classifiers on the Labeled Faces in the Wild data set. The classifiers that were used are listed below. To see all of the code in use for each classifier along with the outputs as a Jupyter Notebook, visit `https://goo.gl/EHGdsw`.

- Support Vector Machine Classifier
- Voting Classifier
- Multilevel Perceptron Classifier (a feed-forward neural network)
- Logistic Regression Classifier
- Stochastic Gradient Descent Classifier
- Gradient Boosting Classifier
- AdaBoost Classifier.

All of these classifiers were trained on the same data set that was prepared the exact same way. The only difference was due to the fact that choosing the images in the training and test set was stochastic and was different for each classifier (and in fact, different for multiple runs of the same classifier). The advantage of this is for direct comparison of the classifiers.

3.3.2 Results

The results for the various Scikit-Learn classifiers are found in Table 1. There were not too many surprises with the results. Since these classifiers are using the output of Primary Components Analysis (PCA) and not a CNN, the results should not be expected to be as good as the state-of-the-art results. It is surprising, however, that the AdaBoost classifier did as poorly as it did. Perhaps there are some adjustments of the parameters or data preparation that could improve the performance.

```
./demos/classifier.py train ./generated-embeddings/
```

Figure 9. Command line instruction to train a Support Vector Machine on the generated embeddings

```
for fn in testing-images/*; do ./demos/classifier.py infer
 ↪  ./generated-embeddings/classifier.pkl $fn 2> /dev/null; done
```

Figure 10. Command line instruction used to classify all the testing images in OpenFace

Figure 11. An example image labeled michael-lane

Figure 12. An example image labeled rebecca-brown

Figure 28 is the confusion matrix from the SVM classifier. Clearly, the results are not up to the same standards as modern face recognition systems like the ones used by

Figure 13. An example image labeled william-lane

Classifier	Precision	Recall	F_1	Accuracy
SVM	0.86	0.86	0.86	0.86
Voting	0.86	0.86	0.86	0.86
MLP	0.84	0.84	0.84	0.84
Logistic Reg	0.82	0.82	0.81	0.82
SGD	0.77	0.78	0.77	0.78
Gradient Boost	0.67	0.65	0.62	0.65
AdaBoost	0.52	0.52	0.51	0.52

Table 1. Results of Scikit-Learn classifiers

OpenFace or (presumably) AWS Rekognition. However, the results are reasonably good and might suffice for some applications.

Finally, Figure 29 depicts an example of the results of the Scikit-Learn classifiers used for this paper. The left side shows the face being evaluated, the predicted class, and the ground truth value. Most predictions were correct in the ex-

Figure 14. An example image labeled joshua-lane

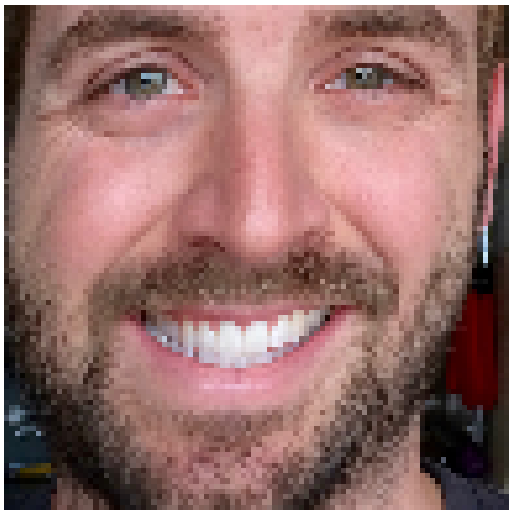

Figure 15. The michael-lane image after alignment

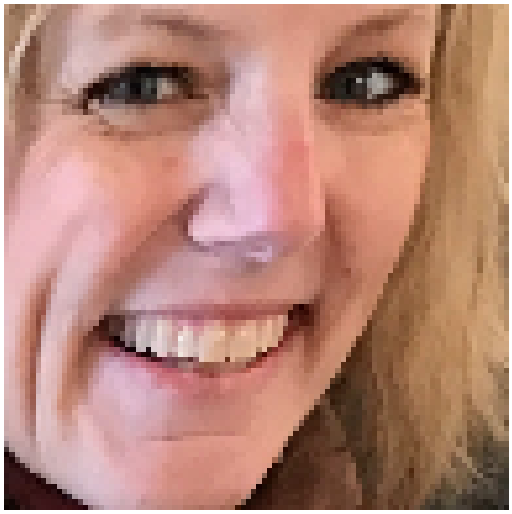

Figure 16. The rebecca-brown image after alignment

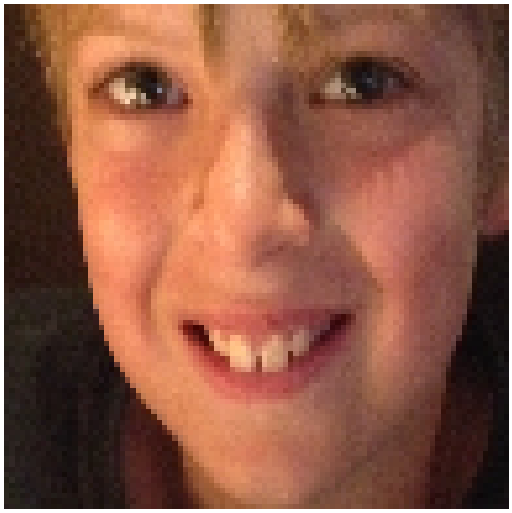

Figure 17. The william-lane image after alignment

ample shown. In fact, only one of the images in the gallery is misclassified. The right side of the figure displays the eigenface representations of the various faces. Humans who are familiar with the faces being considered in the Labeled Faces in the Wild dataset, can determine what faces many of the eigenfaces represent.

4. Conclusions

Since face detection and recognition is a classic problem of computer vision, so it is not surprising that there are so many ways of easily implementing face detection and recognition. The options available to developers vary from free, older methods such as those employed by Scikit-Learn to free modern methods such as those employed by OpenFace to paid state-of-the-art systems such as AWS Rekognition. No face recognition system is perfect and there are still scenarios where humans can easily identify a face that cannot be identified by even the most modern systems.

Although face detection and recognition is not quite up to the standard of human face detection and recognition, modern face recognition is more than adequate when viewed as a tool to help us accomplish many extremely useful tasks.

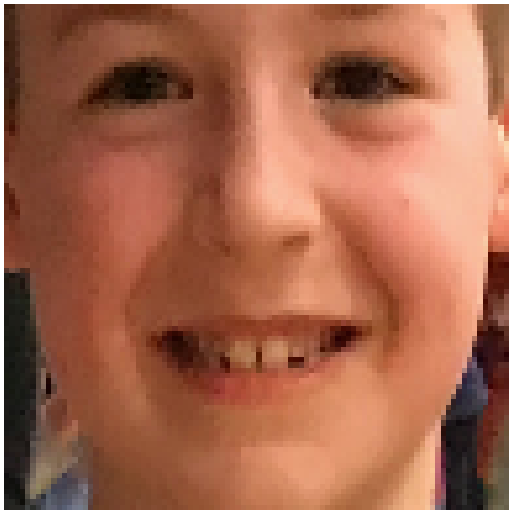

Figure 18. The joshua-lane image after alignment

Figure 19. Joshua-Lane-3.jpg: Face paint, no problem

Figure 20. Michael-Lane-1.jpg: Strange headgear, side angle, no problem

Figure 21. Rebecca-Brown-1.jpg: small, hair-covered, no problem

References

[1] B. Amos, B. Ludwiczuk, and M. Satyanarayanan. Openface, 2016.

[2] B. Amos, B. Ludwiczuk, and M. Satyanarayanan. Openface: A general-purpose face recognition library with mobile applications. Technical report, CMU-CS-16-118, CMU School of Computer Science, 2016.

[3] G. Bradski. *Dr. Dobb's Journal of Software Tools*, 2000.

[4] A. Geitgey. Machine learning is fun! part4, 2016.

[5] G. B. Huang, M. Ramesh, T. Berg, and E. Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst, October 2007.

[6] M. Matusugu, K. Mori, Y. Mitari, and Y. Kaneda. Subject independent facial expression recognition with robust face detection using a convolutional neural network. 2003.

[7] A. Nguyen. Synthesizing the preferred inputs for neurons in neural networks via deep generator networks. *30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain*, 2016.

[8] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss,

Figure 22. William-Lane-4.jpg: Dark shadows, sunglasses, fail

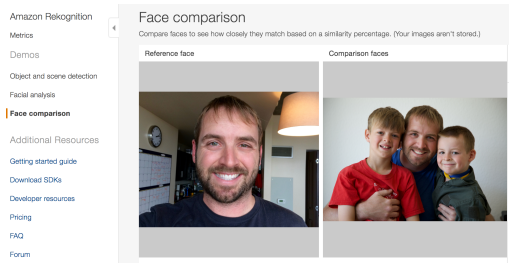

Figure 23. Rekognition demo input

V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[9] S. Tiwari. Face recognition with python, in under 25 lines of code, 2014.

[10] M. Turk and A. Pentland. Face recognition using eigenfaces. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition.*, 1991.

[11] Viola and Jones. Rapid object detection using a boosted cascade of simple features. 2001.

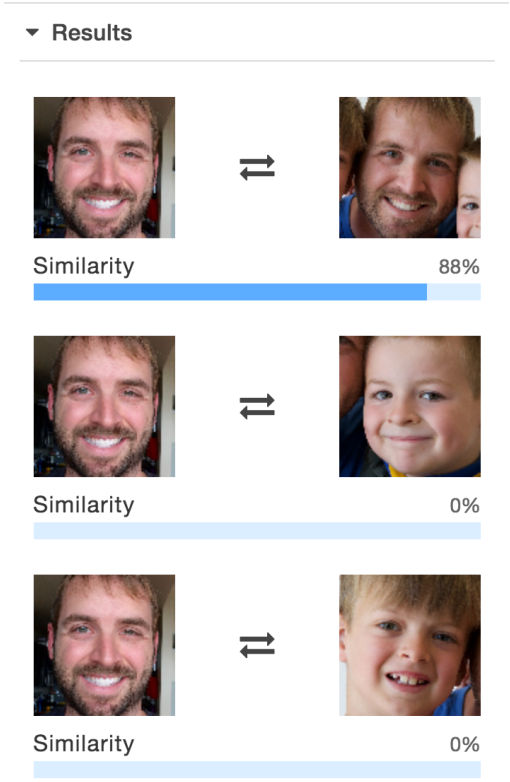

Figure 24. Rekognition demo output

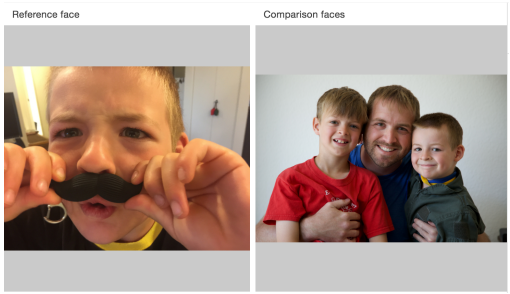

Figure 25. Rekognition occlusion demo

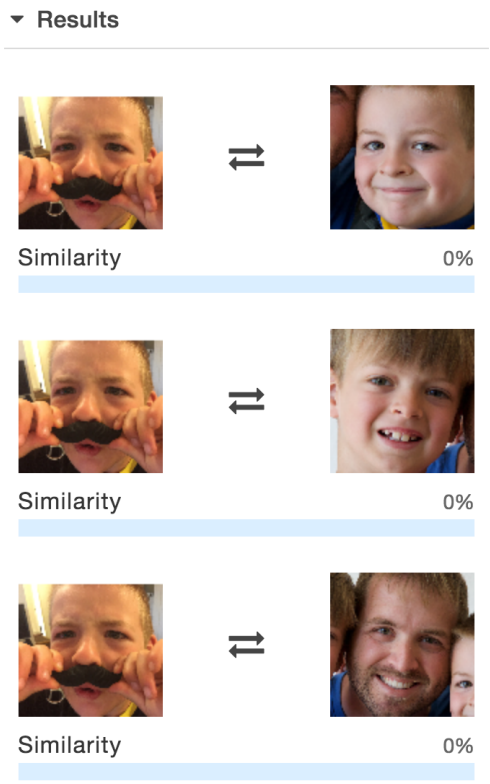

Figure 26. Rekognition occlusion failure result

Figure 27. Examples of the faces found in the LFW dataset

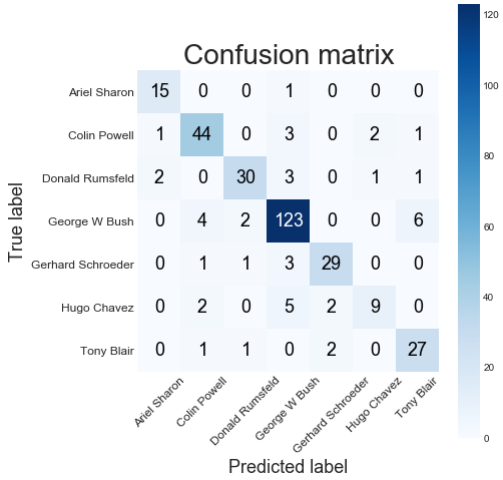

Figure 28. Confusion Matrix for the SVM Classifier

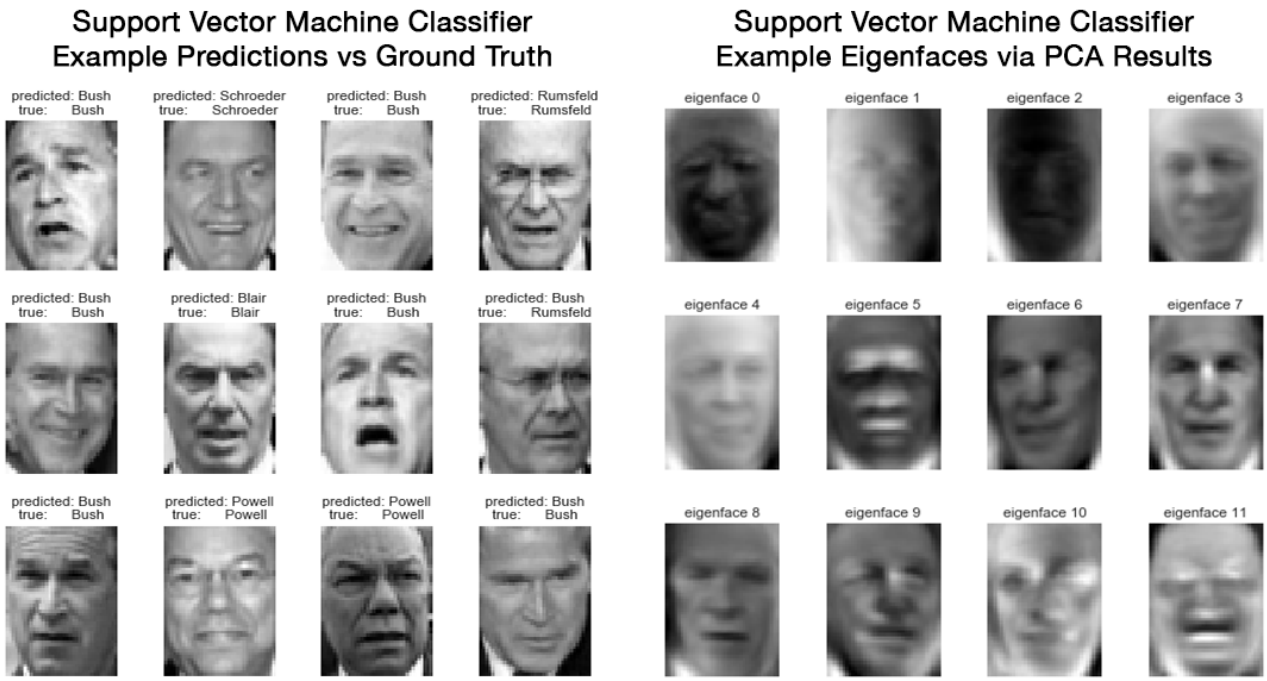

Figure 29. Example results. These were generated by the SVM Classifier